<알고리즘 요약 정리본>

4장 선택 알고리즘 (170403 to 170405)

여러분이 지금 갖고 싶은 것이 무엇인가? 필자는 지금 가지고 있는 컴퓨터에 i5 카비레이크 세대(현재 i3 스카이레이크 쓰고 있어서...)로 교체 하는 동시에 메모리를 16기가로 올리는 것을 목표로 돈을 모으고 또 모으고 있다. 하지만 필자도 돈을 더 모으게 된다면 이런 것들만이 아니라 더더욱 갖고 싶은 것이 언젠가는 분명 생기게 된다. 이처럼 데이터들이 점차 많아지면서 적합한 선택을 위해서 이번에는 선택 알고리즘에 대해서 공부를 해 볼 필요가 있다. 그래서 이번에는 선택에 관련된 알고리즘을 공부해보는 기회를 가져보자.

1. 선택 알고리즘(Selection Algorithm)

예를 들어 초딩 때나 유치원 때 여러 친구들 중에서 '누가 제일 좋아?'를 물어본 적이 있을 것이다. 그럼 여러 친구들 중에서 '나는 철수가 좋아, 그리고 유리는 두 번째로 좋아. 그리고 나는 훈이가 제일 싫어.' 이런 식으로 친구들끼리 내 자신을 위한 호감도를 서열화하는 경우를 생각해봤을 것이다. 이것이 바로 선택 알고리즘의 일상적인 접근으로 살펴볼 수 있겠다. 예를 들어 아래와 같은 데이터 배열들이 있다고 가정을 하자.

29	8	22	4	3	7	20	14	10

그러면 이러한 데이터들 중에서 최댓값을 찾는 방법은 아시다시피 29부터 정리해서 시작하면 29라는 것을 금방 찾을 수 있다. 최댓값을 꺼내는 알고리즘의 시간 복잡도는 O(n)으로 살펴볼 수 있지만, 만일 3번째로 작은 값을 찾으라고 하면 Ω(n)(하한적인 알고리즘 범위 내에서...)으로만은 불충분하다. 적어도 3, 4, 7, 8, ... 이런 식으로 정렬이 되었으면 쉽게 골라낼 수 있는데 정렬 알고리즘을 이용해서 접근을 해야 하니 O(n log n)보다 느릴 수가 없다.(물론 기수 정렬처럼 O(n)으로 끝낼 수 있는 마법의 정렬 알고리즘이 있지만, 이의 공간 복잡도는 O(n)이니...) 그래서 공간적으로도 효율적이고, 시간적으로도 효율적인 알고리즘에 대해 알아볼 필요가 있다.

1-1) 평균 선형시간 선택 알고리즘의 원리

우리가 공간 복잡도가 O(1)이고 시간 복잡도가 O(n log n)을 만족하는 정렬 알고리즘이 바로 퀵 정렬(Quick Sort)이다. 퀵 정렬에 대해서 자세히 모르겠다면 4장 고급정렬 요약정리를 참조하면 도움 되겠다. 방금 문제에서 요구했던 3번째로 작은 값을 찾아보는 원리를 그림으로 설명해 보겠다.

29	8	22	4	3	7	20	14	10

우선 기본적으로 기준 값은 마지막에 있는 10이 되시겠다. 그러면 partition을 통해서 아래와 같이 재정의할 수 있다.

8	4	3	7	10	29	20	14	22

아시다시피 작은 값들이 모인 구역(1구역), 큰 값들이 모인 구역(2구역)으로 다시 정의되었다. 여기서 partition의 결과는 4가 나와서 1구역 내에서 3번째로 작은 값들에 대해서 검색을 하게 된다. 그래서 1구역을 살펴보게 된다면...

4	3	7	8	10	29	20	14	22

이처럼 1구역에서도 정리가 이루어지면서 partition은 2가 나오게 된다. 그래서 우리가 3번째로 작은 값인 7에 대해서는 middle_nth=2-0+1로 인해서 3이 나와서 결국에는 7이 나오게 된다. 물론 middle_nth가 5보다 큰 경우에는 29부터 22까지 partition 메소드를 통해서 선택을 하게 된다. 어째보면 퀵 정렬과 비스듬한 원리로 돌아가게 되지만 이에 대한 알고리즘을 알아보는 기회를 가져보자.

1-2) 평균 선형시간 선택 알고리즘

```
public static void swap(int[] arr, int x, int y){
    int temp=arr[x];
    arr[x]=arr[y];
    arr[y]=temp;
}
public static int partition(int[] arr, int start, int end){
    int j=start-1;
    int compare=arr[end];
    for(int k=start;k<=end-1;k++){
      if(a[k]<compare)
        swap(arr, ++j, k);
    }
    swap(arr, j+1, end);
    return j+1;
}
public static int select(int[] arr, int start, int end, int nth){
    if(start>=end) return arr[start];
    int pivot=partition(arr, start, end);
    int middle_nth=pivot-start+1;

    if(nth=middle_nth) return a[pivot];
    if(nth<middle_nth)
      return select(arr, start, pivot-1, nth);
    else
      return select(arr, pivot+1, end, nth-middle_nth);
}
```

퀵 정렬에 본래 존재하였던 메소드인 partition 메소드를 이용해서 값이 작은 구역은 1구역 내에서 다시 비교, 값이 큰 구역은 2구역 내에서 다시 비교하게 해주는 메소드로도 볼 수 있다. 예를 들어 방금 배열을 통해서 2번째로 작은 값을 찾는 과정과 6번째로 작은 값을 찾는 과정을 사례로 설명을 들어가겠다.

Case 1) 2번째로 작은 값을 찾는 방법

29	8	22	4	3	7	20	14	10

1> 위와 같은 배열로 select(arr, 0, 8, 2)로 접근을 해보겠다. 우선 nth는 몇 번째로 작은지에 대해서 값을 찾아보는데 쓰이는 매개변수 중의 하나이다.
2> partition을 통해서 기준 값에 대한 인덱스를 골라온다면 아래와 같이 정리될 수 있는데,

8	4	3	7	10	29	20	14	22

이 중에서 기준 값에 대한 인덱스(pivot)가 결국에 4

가 되면서 middle_nth가 pivot-start+1를 통해 4-0+1이 되면서 5가 된다.
3> 그렇지만 nth는 5보다 작게 되어 select(arr, 0, 3, 2)를 재귀적으로 호출하게 된다.
4> 그리고 select(arr, 0, 3, 2)를 통해서 아래와 같이 다시 정리될 수 있다.

8	4	3	7					

7를 기준으로 다시 pivot를 골라내는데 있어서 아래와 같이 정리되면서... 결국 pivot는 2가 되면서 middle_nth가 (2-0+1)이 되면서 3이 된다. 하지만 애석하게도 nth는 middle_nth보다 작아서 select(arr, 0, 1, 2)가 재귀 호출된다.

4	3	7	8					

5> 3을 기준으로 다시 pivot를 골라내는데 아래와 같이 정리가 다시 되면서 pivot는 0이 된다. 그래서 middle_nth가 (0-0+1)이 되면서 결국 1이 된다. 그래서 nth는 middle_nth보다 크게 되어 결국 select(arr, 1, 1, 2)가 실행이 된다.

3	4							

6> 허나 재귀 호출을 하게 되면서 start와 end가 결국 같게 되어서 if(start>=end) 아래부로는 실행되지 않아서 결국에 arr[1]에 대해서 반환을 하게 되어 배열 arr에서 2번째로 작은 값인 2가 반환되면서 상황종료.

Case 2) 6번째로 작은 값을 찾는 방법

29	8	22	4	3	7	20	14	10

1> 앞서서 1번과 달리 select(arr, 0, 8, 6)을 접근해보면, nth는 6이 되면서 아래처럼 다시 정리될 수 있겠다.
방금처럼 middle_nth는 5가 되는데 nth는 6이라서 결국에는 마지막 문장을 통해서 select(arr, 5, 8, (6-5))가 재귀 호출된다.

8	4	3	7	10	29	20	14	22

2> 그럼 아래와 같이 29에서부터 22까지만 비교를 하게 된다.

				29	20	14	22

22가 기준 값이 되면서 pivot는 7이 되면서 middle_nth는 7-5+1이 되면서 3이 나오게 된다. 그렇지만 nth는 middle_nth보다 작게 되면서 select(arr, 5, 6, 1)을 재귀호출하게 된다.

				20	14	22	29

3> 그럼 20과 14에 대한 비교를 들어가게 되면서 14가 기준 값이 되면서 아래와 같이 정리된다. 그래서 pivot는 5가 되면서 middle_nth는 5-5+1를 통해 1이 되면서 결국 middle_nth이랑 nth가 서로 같게 되면서 결국 arr[5]를 반환하는 것으로 종결한다.

					14	20	

최종적으로 6번째로 작은 값(혹은 4번째로 큰 값)을 반환하게 되면 결국 14가 반환됨으로서 상황종료.

1-3) 퀵 정렬이 될까요? 안 될까요?
우선 아래와 같은 소스들을 참조해보면 새로운 사실에 대해서 알 수 있겠다.

```
public static void swap(int[] arr, int x, int y){
    ...(위와 똑같음)...
}
public static int partition(int[] arr, int start, int end){
    ...(위와 똑같음)...
}
public static int select(int[] arr, int start, int end, int nth){
    ...(위와 똑같음)...
}
public static void main(String[] args){
    int[] arr={29, 8, 22, 4, 3, 7, 20, 14, 10};
    System.out.println(Arrays.toString(arr));
// 0
    System.out.printf("%d번째로 작은 값 : %d\n", ?, select(arr, 0, arr.length-1, ?); // 1
    System.out.println(Arrays.toString(arr));
// 2
}
```

혹시 선형시간 선택 알고리즘을 통해서 만일 n번째로 작은 값들을 불러오게 되면서 0번째 출력의 결과는 본래와 같지만, 2번째 출력의 결과에 대해서 생각해본 적이 없었을 것이다. 그렇지만 1의 문장의 물음표 부분을 2와 6으로 하면 어떻게 될까?
Case 1) 2로 하는 경우에
[3, 4, 7, 8, 10, 29, 20, 14, 22]
Case 2) 6으로 하는 경우에
[8, 4, 3, 7, 10, 14, 20, 22, 29]
즉 중요한 것은 n번째로 작은 값들에 대해서 선형시간 선택 알고리즘을 통해서 무조건 퀵 정렬이 되는 것이 절대로 아니라는 점을 알아두라는 점이다. 이에 대해서는 1-2를 통한 분석을 통해서 정리를 하면서 공부를 하면 도움이 되니 참조하도록 하자.

1-4) 평균 선형시간 선택 알고리즘의 수행 시간은?
뭐 생각해보면 퀵 정렬과 판박이처럼 생겼지만, 추가가 된 점은 middle_nth랑 nth라고 생각을 하면 되겠다. 배열의 크기가 n인 경우에 배열의 크기가 1일때까지 나누는데 일반적인 경우는 $\log_2 n$이 되겠다. 그리고 아까 partition 메소드에 대한 시간 복잡도가 O(n)이기 때문에 퀵 정렬처럼 시간 복잡도는 O(n logn)이 되시겠다. 하지만 이는 일반적인 경우이고, 아주 최악의 경우에 대해서 다시 생각해볼 필요가 있다.
쉽게 생각해보면 연속적으로 최솟값을 기준값을 고른다면...(즉 쉽게 이야기하면 내림차순으로 정렬된 경우...) 크기가 1일 때까지 나눈 경우에 결국 시간 복잡도가 O(n)이 나오게 되면서 평균 선형시간 선택 알고리즘의 최악의 수행 시간은 O(n^2)이 될 수도 있다.

그렇지만 평균 선형시간 선택 알고리즘에 대해서 자세하게 생각해볼 필요가 있다. 예를 들어 크기가 1/2씩 n번 줄어드는 경우에 대해서 생각을 해보자.

$$S_n = n + \frac{n}{2} + \frac{n}{2^2} + \frac{n}{2^3} + \dots$$

이처럼 계산을 하는 경우에 아래처럼 정리를 할 수 있겠다.

$$S_n = n \times (\frac{(1-(\frac{1}{2})^m)}{1-\frac{1}{2}}) = 2n(1-(\frac{1}{2})^m) \leqq 2n$$

여기서 숙지를 해야 하는 것은 배열은 $\log_2 n$씩 나눠진다는 사실을 알 수 있다. 그래서 m이 $\log_2 n$이 되는데 여기서 $1-(\frac{1}{2})^{\log n}$이 계속 계산을 하면 1로서 극한이 되기 때문에 결국 2n이랑 비스무리하거나 작게 되는 현상을 발견할 수 있을 것이다. 굳이 정렬 과정을 안 하고도 **평균 선형시간 선택 알고리즘 시간 복잡도는 O(n)도 될 수 있다.**

2. 최악의 경우 선형시간 선택 알고리즘

partition의 결과로 나누어진 배열이 크기를 등비수열로 줄어들게끔 하기 위해서는 어떻게 해야 될까? 바로 중앙 값을 찾으는 것을 목표로 잡아야겠다. 그러면 배열의 크기를 1/2씩 줄이면서 잡는 것이 올바른 방법이지만, 물론 1/3로 줄여도, 1/10으로 줄여도... 등비수열로 줄이기만 하면 O(n)으로 충분히 만들 수 있지만, 이는 최악의 경우(내림차순의 경우)를 고려하지 않았으므로 O(n^2)도 나올 수가 있는 것이다. 그렇지만 최악의 경우 선형시간 선택 알고리즘을 방지하기 위한 알고리즘이 있다. 그렇지만 실질적으로 작성하는데 있어서 꽤나 난해한 수준이다. 그래서 이 알고리즘에 대한 설명으로 접근을 들어가보겠다.

```
linearSelect (A, start, end, i){                    // 배열 A[start ... end]에서 i번째 작은 원소를 찾는다
    ① 원소의 총 수가 5개 이하이면 원하는 원소를 찾고 알고리즘을 끝낸다. 그 배열을 정렬하고(quick sort), i 번째 값을 찾는다.
    ② 배열을 5개 크기 배열로 그룹을 나눈다. 나뉘어진 각 그룹의 크기는 5 이지만, 마지막 그룹은 5보다 작을 수 있다. 배열의 크기는 n 이라고 할 때, 그룹의 수는 n/5 이다. (n/5 결과에서 소수점 올림)
    ③ 각 그룹에서 중앙값을 (원소가 5개이면 3번째 원소) 찾는다. 각 배열을 정렬해서 중앙값을 찾는다. 이렇게 찾은 중앙값들을 m1,m2,..., m n/5 이라 하자. 이 중앙값을 배열로 만들자. 이 배열을 A1 이라고 하자.
    ④ m1,m2,..., m n/5들의 중앙값을 구한다. 즉 A1 배열에서 A1.length/2 번째 값을 찾자. M = linearSelect(A1, 0, A1.length-1, A1.length/2) 재귀호출
    ⑤ 위에서 찾은 M 값을 기준원소로 삼아 A 배열을 분할한다. (M 값보다 작거나 같은 것은 M의 왼쪽에, M 값보다 큰 것은 M의 오른쪽에 오도록)
    ⑥ 위에서 찾은 M 값의 위치 인덱스를 j 라고 할 때, M 값은 A[start..end] 배열에서 j-start+1 번째 작은 값이다.
    if (i == j-start+1) return M;
else if (i < j-start+1) linearSelect(A, start, j-1, i);
else linearSelect(A, j+1, end, i - (j-start+1));
}
```

linearSelect 알고리즘을 딱 살펴봐도 무슨 이야기인지 잘 모를 것이다. 그렇지만 여기서 알아둬야 하는 점은 배열의 크기를 비교하는데 있어서 등차수열로 비교하는 개념(방금 전처럼 내림차순처럼 최악의 상황인 경우에 최솟값을 pivot로 때려잡는 경우)이 아닌 등비수열로 비교하는 것을 목표(pivot를 기준으로 해서 반반씩 나누어 가지어 공평성을 제공)로 두고 숙지하는 점을 포인트로 잡는다.

우선 배열이 아래와 같이 방대하게 구성되어 있다고 가정을 해보자. 위의 알고리즘을 통해서 어떠한 원리로 돌아가는지에 대해 알아볼 필요가 있으며 알고리즘을 통해서 돌아가는 원리를 알아보는 것만으로 마무리를 짓는 것으로 하자.(본래 이 알고리즘을 구축하기에도 너무 어렵기 때문이다...ㅜㅜ)
{20, 1, 19, 2, 18, 3, 17, 4, 16, 5,
15, 6, 14, 7, 13, 8, 12, 9, 11, 10}

2-1) 알고리즘 작동 원리

linearSelect에 나와 있는 그대로 어떠한 원리로 돌아가는지에 대해서만 간략하게 정리를 해뒀으니 더욱 자세히 공부해보고 싶은 분은 책이나 강의 노트를 참조하는 것이 좋겠다.

① 원소의 수는 총 20개나 되기 때문에 이에 대해서는 씹고 넘어가게 된다.
② 위와 같은 배열을 통해서 5개씩 나누어본다면...
(1그룹)[20, 1, 19, 2, 18]
(2그룹)[3, 17, 4, 16, 5]
(3그룹)[15, 6, 14, 7, 13]
(4그룹)[8, 12, 9, 11, 10]
이렇게 4개의 그룹으로 나뉠 수가 있다. 이제 3번을 통해서 중앙값들을 저장하는 A1 배열을 작성해보면...
③ A1=[18, 5, 13, 10] 이렇게 작성이 되시겠다. 그렇지만 이 배열도 4번의 과정을 통해서 linearSelect(A1, 0, 3, 2)가 재귀 호출이 되면서 따로따로 분해가 되겠다. A1의 길이의 절반인 인덱스를 골라오게 하면서 **배열의 크기를 등비수열로 줄여나가는 과정을 통해서 값을 선택하는 과정이 여기서부터 시작된다고 보면 무난하겠다.**
④ A1의 길이는 4가 나오게 됨으로서 2번째 인덱스인 13이 나오게 된다. 그러면 13이 기준 원소로 삼게 되니깐 5번처럼 새로이 정리될 수 있겠다.
⑤
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 11] 13
[20, 19, 18, 17, 16, 15, 14]
⑥ 그럼 이를 통해서 13보다 작은 값은 13 이전에 값들로, 13보다 큰 값은 13 다음 값으로 찾는 원리인데 이를 또 linearSelect란 메소드를 통해서 재귀 호출을 하는 원리이다.
여기서 선택 알고리즘의 수행 시간은 O(n)으로 정리되는데 이에 대해서 강의 노트를 보는 참조하거나 책을 참조해보면서 공부를 해보길 바란다. 필자도 다음 장에 대해서 자세히 공부를 해야 되니 이에 대해서는 알고리즘을 심도있게 공부할 수 있는 기회가 되면 다시 공부해보는 것으로 하자.(쉽게 이야기해서 이러한 알고리즘이 있구나 라고 생각하고 일단 넘어가도록 하자.)